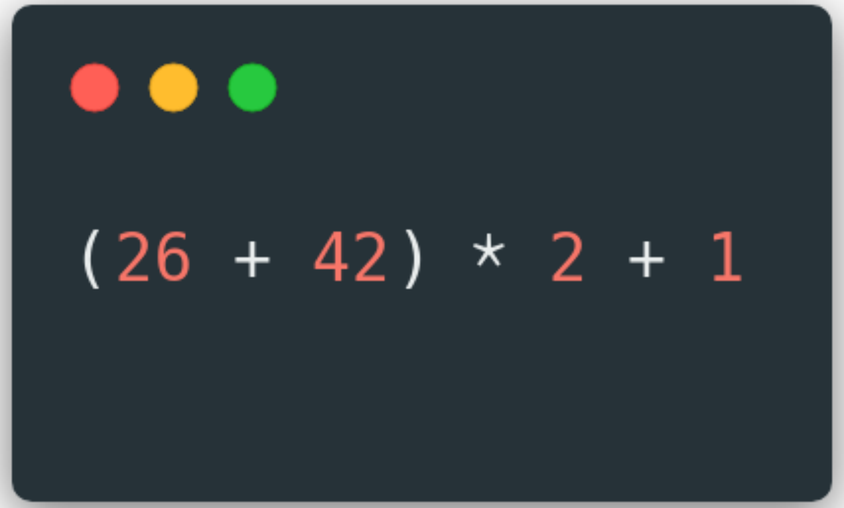
CMPU 240 · Theory of Computation

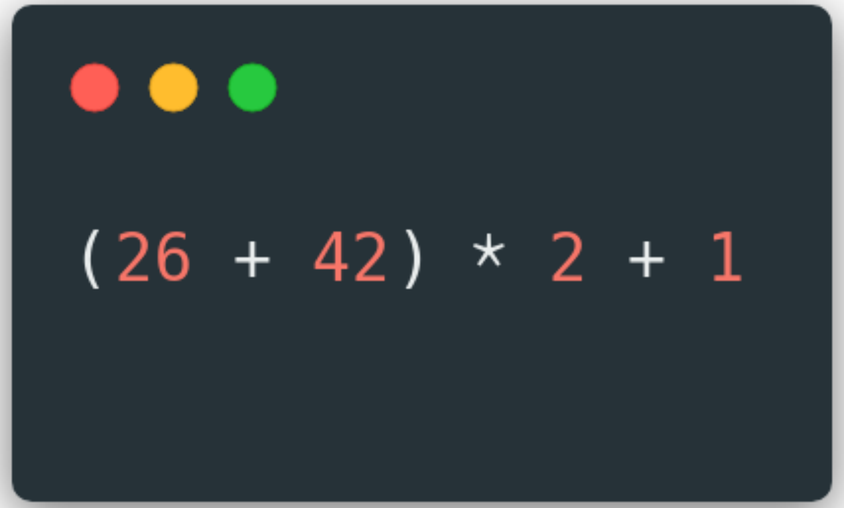
Context-free Grammars

4 October 2022

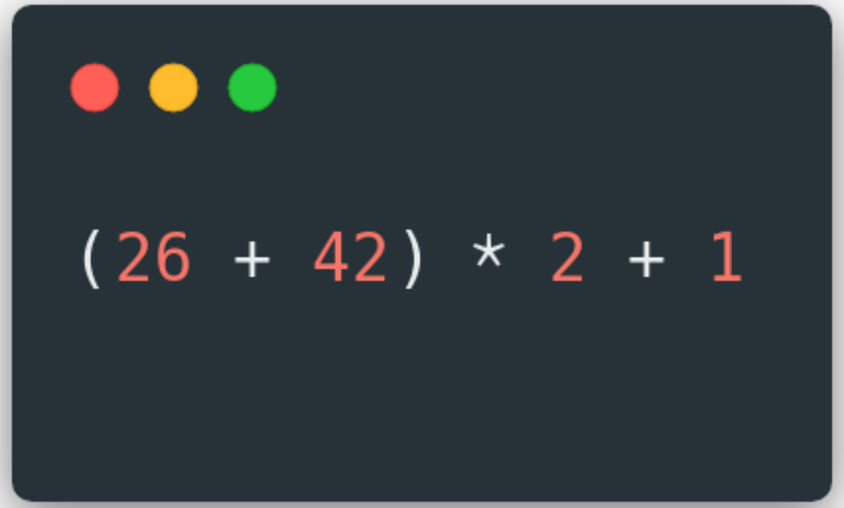
“Many of us have an innate predisposition to be curious. I believe that after a traditional education, or working in an environment with many people, curiosity is a decision requiring intent and discipline.”

Jony Ive, *The Wall Street Journal Magazine*, 4 October 2021


(26 + 42) * 2 + 1



When I'm programming in Python, how does my computer know what this sequence of characters means?



When I'm programming in Python, how does my computer know what this sequence of characters means?

How can it determine whether or not this expression is even syntactically valid?

Thousands of TREES (PLURAL NOUN) ago, there were calendars that enabled the ancient COMPUTERS (PLURAL NOUN) to divide a year into twelve COOKIES (PLURAL NOUN), each month into 25 (NUMBER) weeks, and each week into seven TOMATOES (PLURAL NOUN). At first, people told time by a sun clock, sometimes known as the CHOCOLATE (NOUN) dial. Ultimately, they invented the great timekeeping devices of today, such as the grandfather PUMPKIN (NOUN), the pocket CHEESE (NOUN), the alarm MILK (NOUN), and, of course, the EYE (PART OF BODY) watch. Children learn about clocks and time almost before they learn their AB M (LETTER OF THE ALPHABET)'s. They are taught that a day consis

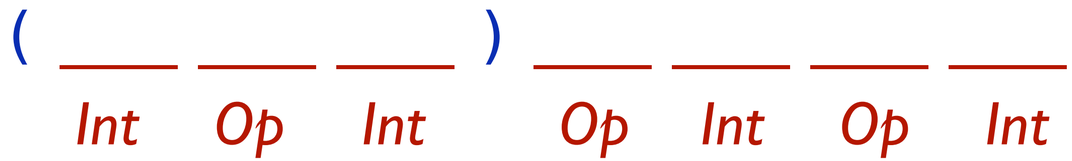
KIDS (PLURAL NOUN), an hour has 60 BATS (PLURAL NOUN), and a minut

WITCHES (PLURAL NOUN). By the time they are in kindergarten, they

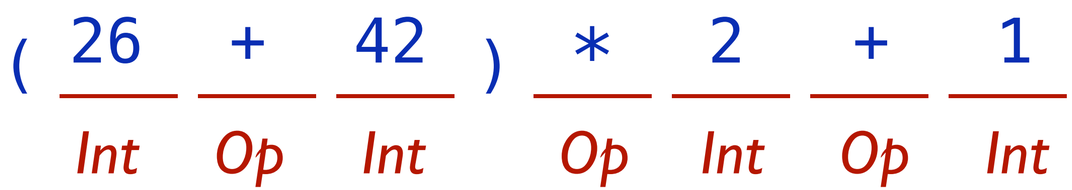
the big TOE (PART OF THE BODY) is at twelve and the little NOSE (PART OF THE BODY) is at three, that it is 13 (NUMBER) o'clock. I wish we could continue this

In Mad Libs, you have blanks that you fill with an indicated type of word.

We can describe languages in a similar way:

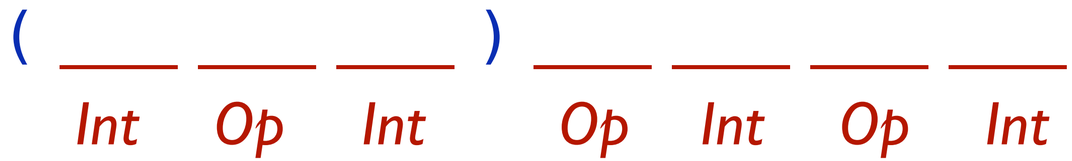


We can describe languages in a similar way:

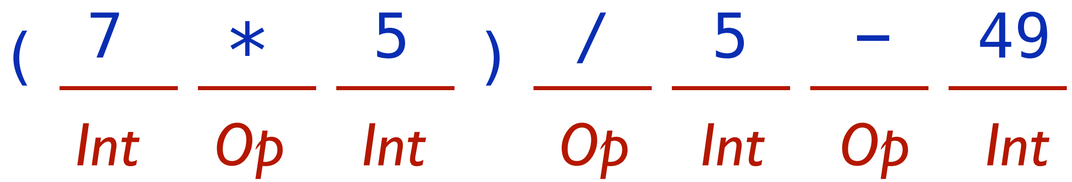


This is one way to fill in the blanks.

We can describe languages in a similar way:

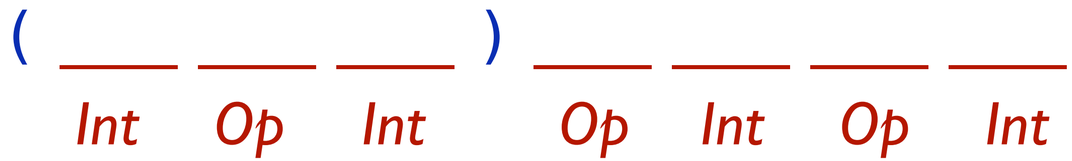


We can describe languages in a similar way:



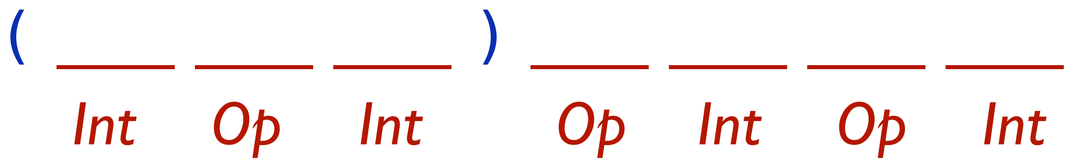
This is another!

If you had a computer pre-programmed with a template like this, then you could enter a string and check whether it's valid.



You can also understand what individual pieces of the string mean based on which part of the template they're filling in!

But this only lets us make one form of arithmetic expression. What about others?



Recursive Mad Libs

Expr

Recursive Mad Libs

Expr

What can an arithmetic expression be?

Recursive Mad Libs

`int`

Expr

What can an arithmetic expression be?

`int`	A single number

Recursive Mad Libs

Expr

What can an arithmetic expression be?

`int`	A single number

Recursive Mad Libs

Expr

What can an arithmetic expression be?

`int`	A single number
Expr Op Expr	Two expressions joined by an operator

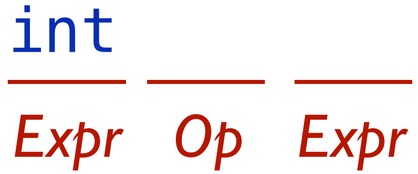
Recursive Mad Libs

Expr *Op* *Expr*

What can an arithmetic expression be?

`int`	A single number
Expr Op Expr	Two expressions joined by an operator

Recursive Mad Libs



What can an arithmetic expression be?

`int`	A single number
Expr Op Expr	Two expressions joined by an operator

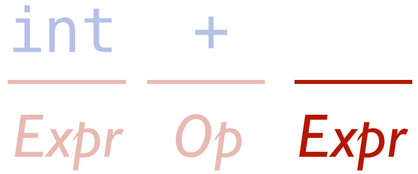
Recursive Mad Libs

`int`	`+`	
Expr	*Op*	*Expr*

What can an arithmetic expression be?

`int`	A single number
Expr Op Expr	Two expressions joined by an operator

Recursive Mad Libs



What can an arithmetic expression be?

`int`	A single number
Expr Op Expr	Two expressions joined by an operator

Recursive Mad Libs

What can an arithmetic expression be?

`int`	A single number
Expr Op Expr	Two expressions joined by an operator

Recursive Mad Libs

What can an arithmetic expression be?

`int`	A single number
Expr Op Expr	Two expressions joined by an operator

Recursive Mad Libs

`int`	`+`	`int`	___	___
Expr	*Op*	*Expr*	*Op*	*Expr*

What can an arithmetic expression be?

`int`	A single number
Expr Op Expr	Two expressions joined by an operator

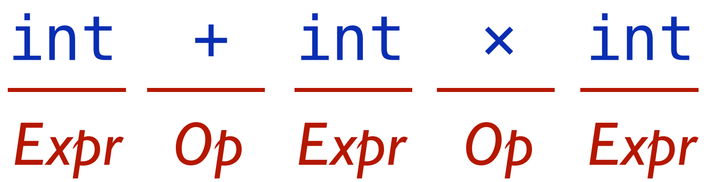
Recursive Mad Libs

`int`	+	`int`	×	___
Expr	*Op*	*Expr*	*Op*	*Expr*

What can an arithmetic expression be?

`int`	A single number
Expr Op Expr	Two expressions joined by an operator

Recursive Mad Libs



What can an arithmetic expression be?

`int`	A single number
Expr Op Expr	Two expressions joined by an operator

Recursive Mad Libs

Expr

What can an arithmetic expression be?

`int`	A single number
Expr Op Expr	Two expressions joined by an operator

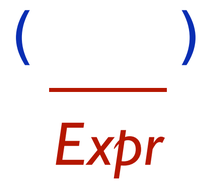
Recursive Mad Libs

Expr

What can an arithmetic expression be?

`int`	A single number
Expr Op Expr	Two expressions joined by an operator
(*Expr*)	A parenthesized expression

Recursive Mad Libs



What can an arithmetic expression be?

`int`	A single number
Expr Op Expr	Two expressions joined by an operator
(*Expr*)	A parenthesized expression

Recursive Mad Libs

What can an arithmetic expression be?

`int`	A single number
Expr Op Expr	Two expressions joined by an operator
(*Expr*)	A parenthesized expression

Recursive Mad Libs

(____ ____ ____)

Expr *Op* *Expr*

What can an arithmetic expression be?

`int`	A single number
Expr Op Expr	Two expressions joined by an operator
(*Expr*)	A parenthesized expression

Recursive Mad Libs

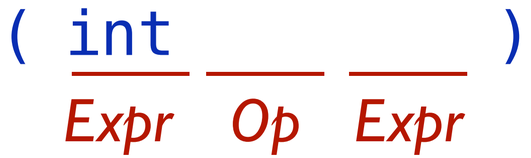
(____ ____ ____)

Expr *Op* *Expr*

What can an arithmetic expression be?

`int`	A single number
Expr Op Expr	Two expressions joined by an operator
(*Expr*)	A parenthesized expression

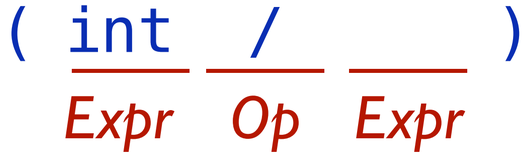
Recursive Mad Libs



What can an arithmetic expression be?

`int`	A single number
Expr Op Expr	Two expressions joined by an operator
(*Expr*)	A parenthesized expression

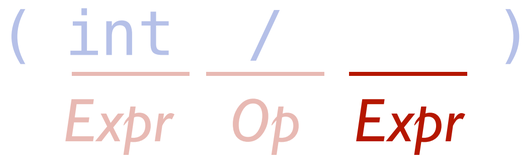
Recursive Mad Libs



What can an arithmetic expression be?

`int`	A single number
Expr Op Expr	Two expressions joined by an operator
(*Expr*)	A parenthesized expression

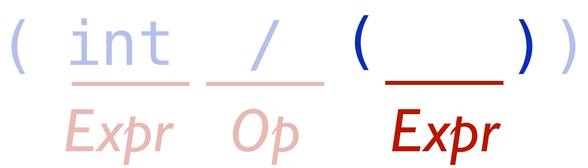
Recursive Mad Libs



What can an arithmetic expression be?

`int`	A single number
Expr Op Expr	Two expressions joined by an operator
(*Expr*)	A parenthesized expression

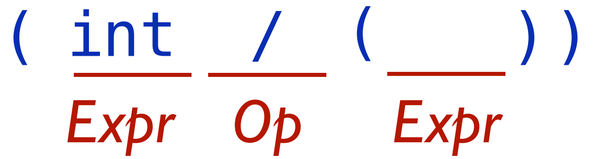
Recursive Mad Libs



What can an arithmetic expression be?

`int`	A single number
Expr Op Expr	Two expressions joined by an operator
(*Expr*)	A parenthesized expression

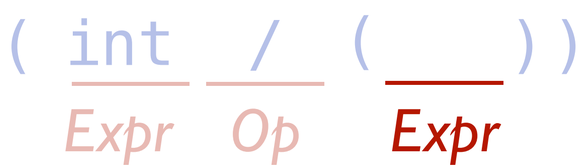
Recursive Mad Libs



What can an arithmetic expression be?

`int`	A single number
Expr Op Expr	Two expressions joined by an operator
(*Expr*)	A parenthesized expression

Recursive Mad Libs



What can an arithmetic expression be?

`int`	A single number
Expr Op Expr	Two expressions joined by an operator
(*Expr*)	A parenthesized expression

Recursive Mad Libs

What can an arithmetic expression be?

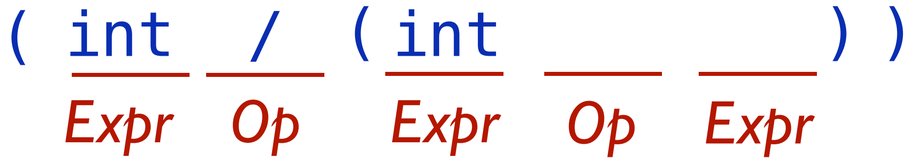
`int`	A single number
Expr Op Expr	Two expressions joined by an operator
(*Expr*)	A parenthesized expression

Recursive Mad Libs

What can an arithmetic expression be?

`int`	A single number
Expr Op Expr	Two expressions joined by an operator
(*Expr*)	A parenthesized expression

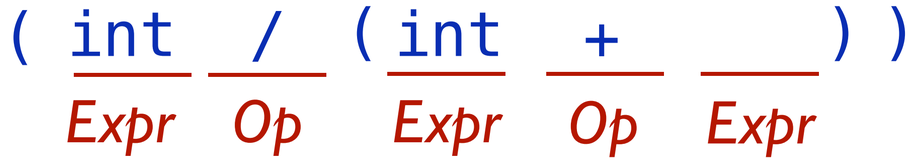
Recursive Mad Libs



What can an arithmetic expression be?

`int`	A single number
Expr Op Expr	Two expressions joined by an operator
(*Expr*)	A parenthesized expression

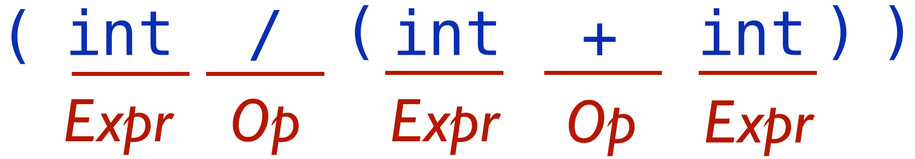
Recursive Mad Libs



What can an arithmetic expression be?

`int`	A single number
Expr Op Expr	Two expressions joined by an operator
(*Expr*)	A parenthesized expression

Recursive Mad Libs



What can an arithmetic expression be?

`int`	A single number
Expr Op Expr	Two expressions joined by an operator
(*Expr*)	A parenthesized expression

Context-free grammars

A *context-free grammar* (CFG) is a way of recursively describing the ways to form the strings in a language.

Here's how we might express the recursive rules we just saw as a CFG:

```
Expr → int
Expr → Expr Op Expr
Expr → (Expr)
  Op → +
  Op → -
  Op → ×
  Op → /
```

Here's how we might express the recursive rules we just saw as a CFG:

Expr → `int`

Expr → *Expr Op Expr*

Expr → `(` *Expr* `)`

Op → `+`

Op → `-`

Op → `×`

Op → `/`

This is called a production rule. It says, "if you see Expr, you can replace it with Expr Op Expr".

Here's how we might express the recursive rules we just saw as a CFG:

Expr → `int`
Expr → *Expr Op Expr*
Expr → `(`*Expr*`)`
Op → `+`
Op → `-`
Op → `×`
Op → `/`

This one says, "if you see Op, you can replace it with –".

Expr → `int`
Expr → *Expr Op Expr*
Expr → `(`*Expr*`)`
Op → `+`
Op → `-`
Op → `×`
Op → `/`

Expr

These red symbols are called nonterminals. They're placeholders that get expanded.

$$
\begin{aligned}
\mathit{Expr} &\rightarrow \texttt{int} \\
\mathit{Expr} &\rightarrow \mathit{Expr}\ \mathit{Op}\ \mathit{Expr} \\
\mathit{Expr} &\rightarrow (\mathit{Expr}) \\
\mathit{Op} &\rightarrow + \\
\mathit{Op} &\rightarrow - \\
\mathit{Op} &\rightarrow \times \\
\mathit{Op} &\rightarrow /
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{Expr} \\
\Rightarrow\ &\mathit{Expr}\ \mathit{Op}\ \mathit{Expr}
\end{aligned}
$$

Expr → `int`
Expr → *Expr Op Expr*
Expr → `(`*Expr*`)`
Op → `+`
Op → `-`
Op → `×`
Op → `/`

Expr
⇒ *Expr Op Expr*
⇒ *Expr Op* `(`*Expr*`)`

The symbols in blue monospace are terminals. They're the final characters used in the string and never get replaced.

$$
\begin{aligned}
\mathit{Expr} &\rightarrow \texttt{int} \\
\mathit{Expr} &\rightarrow \mathit{Expr}\ \mathit{Op}\ \mathit{Expr} \\
\mathit{Expr} &\rightarrow (\mathit{Expr}) \\
\mathit{Op} &\rightarrow + \\
\mathit{Op} &\rightarrow - \\
\mathit{Op} &\rightarrow \times \\
\mathit{Op} &\rightarrow /
\end{aligned}
$$

$$
\begin{aligned}
& \mathit{Expr} \\
\Rightarrow\ & \mathit{Expr}\ \mathit{Op}\ \mathit{Expr} \\
\Rightarrow\ & \mathit{Expr}\ \mathit{Op}\ \texttt{int} \\
\Rightarrow\ & \texttt{int}\ \mathit{Op}\ \texttt{int} \\
\Rightarrow\ & \texttt{int / int}
\end{aligned}
$$

$$
\begin{aligned}
\mathit{Expr} &\rightarrow \texttt{int} \\
\mathit{Expr} &\rightarrow \mathit{Expr}\ \mathit{Op}\ \mathit{Expr} \\
\mathit{Expr} &\rightarrow (\mathit{Expr}) \\
\mathit{Op} &\rightarrow + \\
\mathit{Op} &\rightarrow - \\
\mathit{Op} &\rightarrow \times \\
\mathit{Op} &\rightarrow /
\end{aligned}
$$

$$
\begin{aligned}
& \mathit{Expr} \\
\Rightarrow\ & \mathit{Expr}\ \mathit{Op}\ \mathit{Expr} \\
\Rightarrow\ & \mathit{Expr}\ \mathit{Op}\ (\mathit{Expr}) \\
\Rightarrow\ & \mathit{Expr}\ \mathit{Op}\ (\mathit{Expr}\ \mathit{Op}\ \mathit{Expr}) \\
\Rightarrow\ & \mathit{Expr} \times (\mathit{Expr}\ \mathit{Op}\ \mathit{Expr}) \\
\Rightarrow\ & \texttt{int} \times (\mathit{Expr}\ \mathit{Op}\ \mathit{Expr}) \\
\Rightarrow\ & \texttt{int} \times (\texttt{int}\ \mathit{Op}\ \mathit{Expr}) \\
\Rightarrow\ & \texttt{int} \times (\texttt{int}\ \mathit{Op}\ \texttt{int}) \\
\Rightarrow\ & \texttt{int} \times (\texttt{int} + \texttt{int})
\end{aligned}
$$

Formally, a context-free grammar G is a tuple of four items, (V, Σ, P, S):

V is a set of *nonterminal symbols* (or *variables*),

Σ is a set of *terminal symbols* (the *alphabet*),

P is a set of *production rules* of the form *head* $\rightarrow$ *body*, where

head is a nonterminal and
body is a string of zero or more terminals and/or nonterminals that can be substituted for the head

$S \in V$ is the *start symbol* (which must be a nonterminal) that begins the derivation.

Conventionally, the start symbol is the head of the first production rule.

A notational shorthand

Expr → `int`
Expr → *Expr Op Expr*
Expr → `(`*Expr*`)`
Op → `+`
Op → `-`
Op → `×`
Op → `/`

Expr → `int` | *Expr Op Expr* | `(`*Expr*`)`
Op → `+` | `-` | `×` | `/`

Derivations and parse trees

A *derivation* is a sequence of zero or more steps where nonterminals are replaced by the right-hand side of a production rule.

We write $\alpha \overset{*}{\Rightarrow} \beta$ if string α derives string β by zero or more replacements.

We use Greek letters to stand for strings of terminals and non-terminals.

$$
\begin{aligned}
\mathit{Expr} &\rightarrow \texttt{int} \mid \mathit{Expr}\ \mathit{Op}\ \mathit{Expr} \mid (\mathit{Expr}) \\
\mathit{Op} &\rightarrow \texttt{+} \mid \texttt{-} \mid \texttt{×} \mid \texttt{/}
\end{aligned}
$$

$$
\begin{aligned}
& \mathit{Expr} \\
\Rightarrow\ & \mathit{Expr}\ \mathit{Op}\ \mathit{Expr} \\
\Rightarrow\ & \mathit{Expr}\ \mathit{Op}\ (\mathit{Expr}) \\
\Rightarrow\ & \mathit{Expr}\ \mathit{Op}\ (\mathit{Expr}\ \mathit{Op}\ \mathit{Expr}) \\
\Rightarrow\ & \mathit{Expr}\ \texttt{×}\ (\mathit{Expr}\ \mathit{Op}\ \mathit{Expr}) \\
\Rightarrow\ & \texttt{int}\ \texttt{×}\ (\mathit{Expr}\ \mathit{Op}\ \mathit{Expr}) \\
\Rightarrow\ & \texttt{int}\ \texttt{×}\ (\texttt{int}\ \mathit{Op}\ \mathit{Expr}) \\
\Rightarrow\ & \texttt{int}\ \texttt{×}\ (\texttt{int}\ \mathit{Op}\ \texttt{int}) \\
\Rightarrow\ & \texttt{int}\ \texttt{×}\ (\texttt{int}\ \texttt{+}\ \texttt{int})
\end{aligned}
$$

$$
\mathit{Expr} \overset{*}{\Rightarrow} \texttt{int}\ \texttt{×}\ (\texttt{int}\ \texttt{+}\ \texttt{int})
$$

The language of a grammar

If G is a CFG with alphabet Σ and start symbol S, then the *language of G* is

$$L(G) = \{w \in \Sigma^* \mid S \overset{*}{\Rightarrow}_G w\}.$$

That is, $L(G)$ is the set of strings of terminals derivable from the start symbol.

By $\overset{*}{\Rightarrow}_G$ we're making it explicit that the derivation is using the rules from grammar G.

If G is a CFG with alphabet Σ and start symbol S, then the language of G is $L(G) = \{w \in \Sigma^* \mid S \overset{*}{\Rightarrow}_G w\}$.

Consider the following CFG G over $\Sigma = \{a, b, c, d\}$:

$S \to Sa \mid dT$

$T \to bTb \mid c$

Which of the following strings are in $L(G)$?

dca

dc

cad

bcb

d*T*aa

We have a choice of variable to replace at each step

Derivations may appear different only because we make the *same* replacements in a different order

To avoid such differences, we can restrict the choice.

A *leftmost derivation* always replaces the leftmost variable in a sentential form.

Example

A simple example generates strings of a*s* and b*s* such that each block of a*s* is followed by *at least* as many b*s*

$$S \rightarrow AS \mid \varepsilon$$
$$A \rightarrow aAb \mid Ab \mid ab$$

$$S \to AS \mid \varepsilon$$
$$A \to aAb \mid Ab \mid ab$$

Leftmost derivation:

$$\begin{aligned}
&\underline{S} \\
\Rightarrow\ &\underline{A}S \\
\Rightarrow\ &\underline{A}bS \\
\Rightarrow\ &abb\underline{S} \\
\Rightarrow\ &abb\underline{A}S \\
\Rightarrow\ &abba\underline{A}bS \\
\Rightarrow\ &abbaabb\underline{S} \\
\Rightarrow\ &abbaabb
\end{aligned}$$

Alternative derivation:

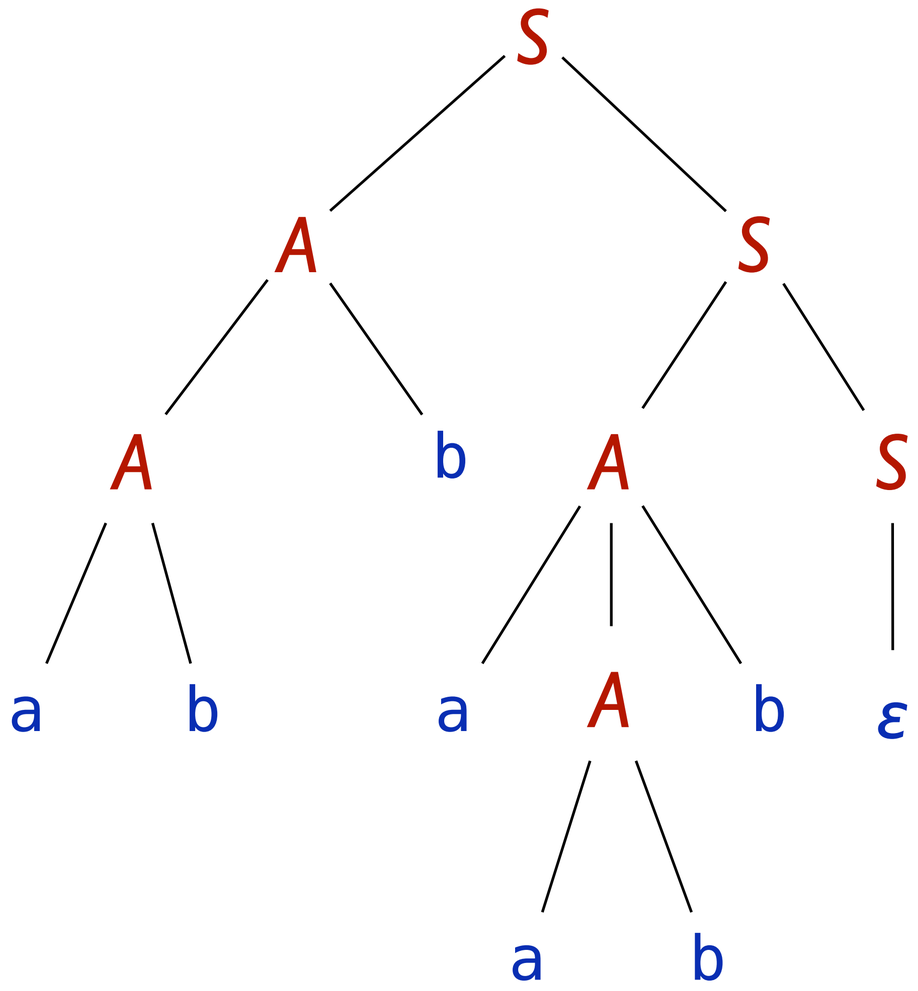
$$\begin{aligned}
&\underline{S} \\
\Rightarrow\ &A\underline{S} \\
\Rightarrow\ &AA\underline{S} \\
\Rightarrow\ &AA\underline{\varepsilon} \\
\Rightarrow\ &A\underline{A} \\
\Rightarrow\ &Aa\underline{A}b \\
\Rightarrow\ &\underline{A}aabb \\
\Rightarrow\ &\underline{A}baabb \\
\Rightarrow\ &abbaabb
\end{aligned}$$

The underlined nonterminal is the one replaced in the next step.

A *parse tree* is a tree encoding which substitutions are in a derivation – but not their order!

$S \rightarrow AS \mid \varepsilon$

$A \rightarrow aAb \mid Ab \mid ab$



Given a context-free grammar, the problem of *parsing* a string is to find a parse tree for that string (if one exists).

Ambiguity

A CFG is called *ambiguous* if there is at least one string that has multiple parse trees.

Equivalently, a CFG is ambiguous if there is at least one string that has multiple leftmost derivations.

Ambiguous grammar: Parse trees

$S \rightarrow AS \mid \varepsilon$

$A \rightarrow Ab \mid aAb \mid ab$

Ambiguous grammar: Parse trees

$S \rightarrow AS \mid \varepsilon$

$A \rightarrow Ab \mid aAb \mid ab$

S

S

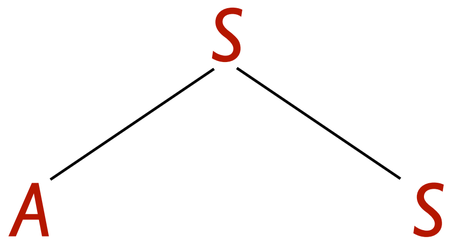
Ambiguous grammar: Parse trees

$S \rightarrow AS \mid \varepsilon$

$A \rightarrow Ab \mid aAb \mid ab$

$\underline{S}$

$\Rightarrow \underline{A}S$



Ambiguous grammar: Parse trees

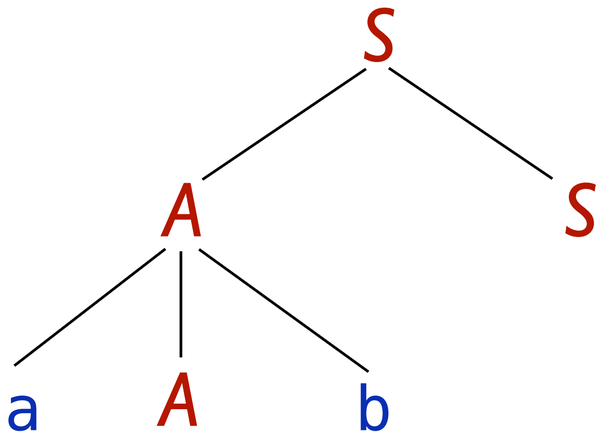
$S \rightarrow AS \mid \varepsilon$

$A \rightarrow Ab \mid aAb \mid ab$

$\underline{S}$

$\Rightarrow \underline{A}S$

$\Rightarrow aA\underline{b}S$



Ambiguous grammar: Parse trees

$S \rightarrow AS \mid \varepsilon$

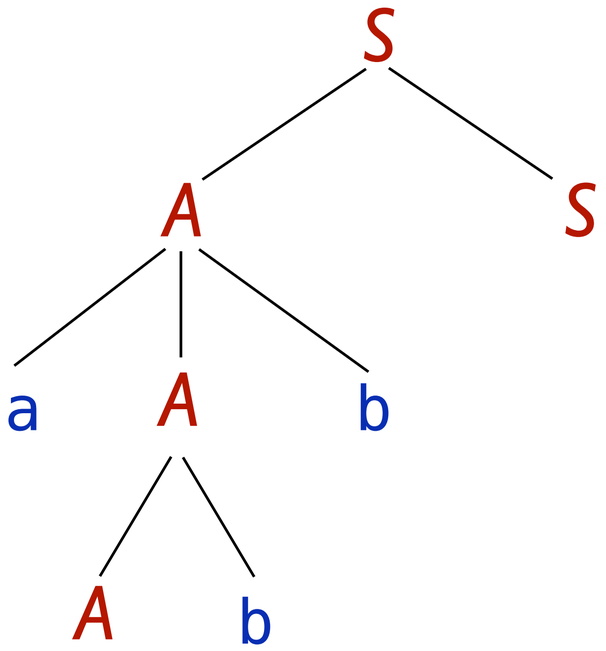
$A \rightarrow Ab \mid aAb \mid ab$

$\underline{S}$

$\Rightarrow \underline{A}S$

$\Rightarrow a\underline{A}bS$

$\Rightarrow a\underline{A}bbS$



Ambiguous grammar: Parse trees

$S \rightarrow AS \mid \varepsilon$

$A \rightarrow Ab \mid aAb \mid ab$

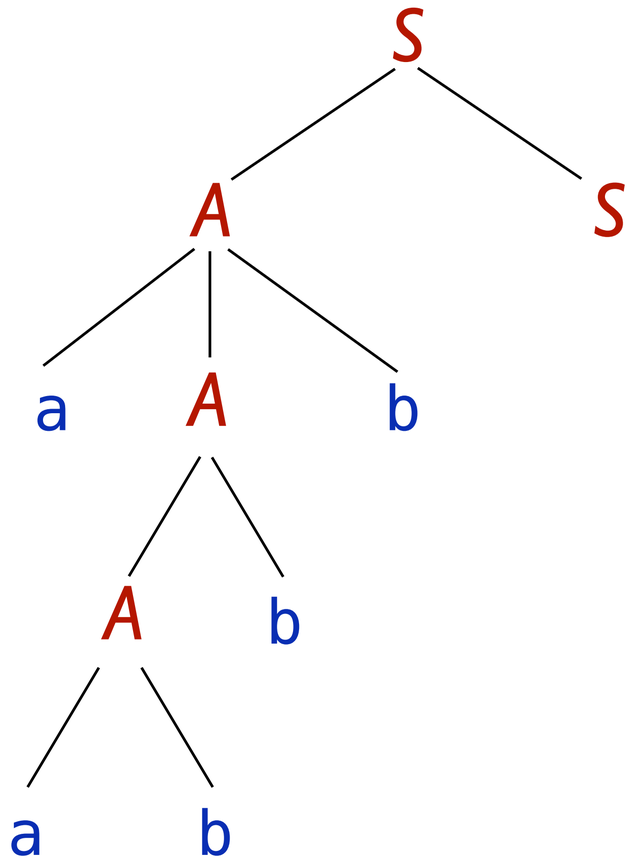
$\underline{S}$

$\Rightarrow \underline{A}S$

$\Rightarrow aA\underline{b}S$

$\Rightarrow aA\underline{b}bS$

$\Rightarrow aabbb\underline{S}$



Ambiguous grammar: Parse trees

$S \rightarrow AS \mid \varepsilon$

$A \rightarrow Ab \mid aAb \mid ab$

$\underline{S}$

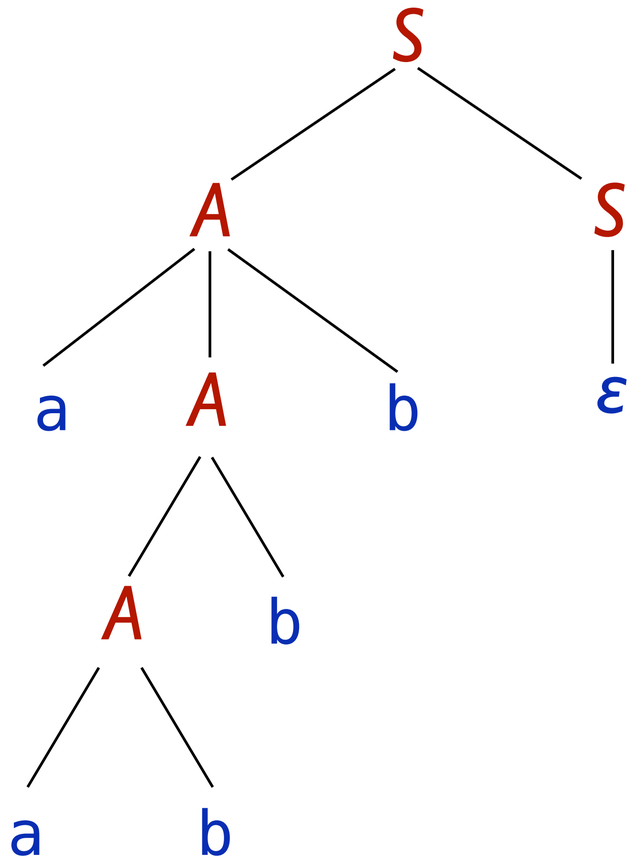
$\Rightarrow \underline{A}S$

$\Rightarrow aA\underline{b}S$

$\Rightarrow aA\underline{b}bS$

$\Rightarrow aabbb\underline{S}$

$\Rightarrow aabbb$



Ambiguous grammar: Parse trees

$S \rightarrow AS \mid \varepsilon$

$A \rightarrow Ab \mid aAb \mid ab$

$\underline{S}$

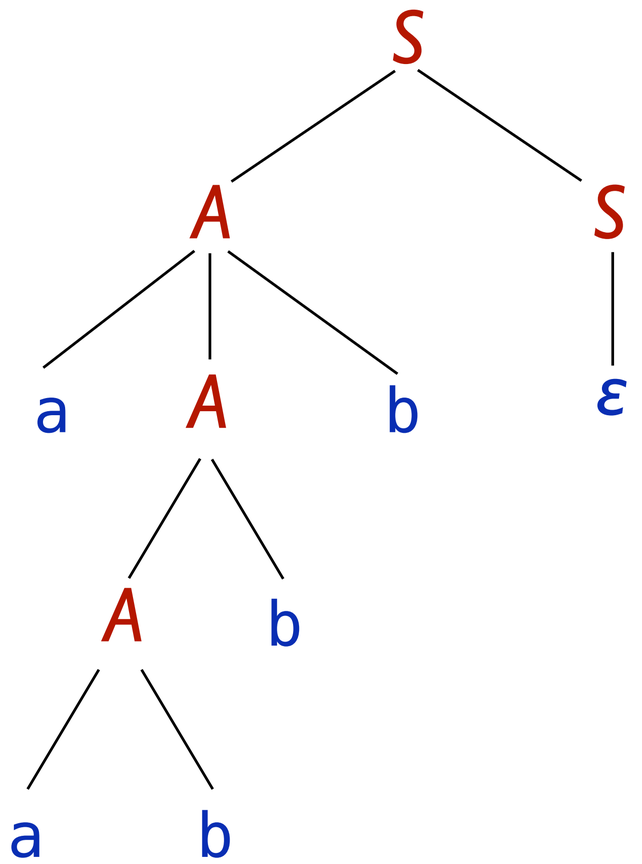
$\Rightarrow \underline{A}S$

$\Rightarrow aA\underline{b}S$

$\Rightarrow a\underline{A}bbS$

$\Rightarrow aabbb\underline{S}$

$\Rightarrow aabbb$



$\underline{S}$

S

Ambiguous grammar: Parse trees

$S \rightarrow AS \mid \varepsilon$

$A \rightarrow Ab \mid aAb \mid ab$

$\underline{S}$

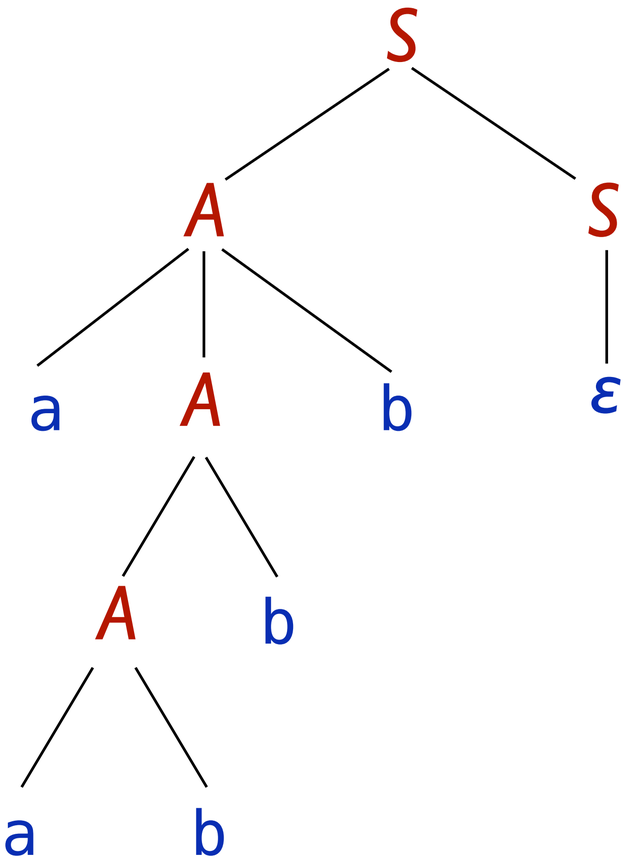
$\Rightarrow \underline{A}S$

$\Rightarrow aA\underline{b}S$

$\Rightarrow aA\underline{b}bS$

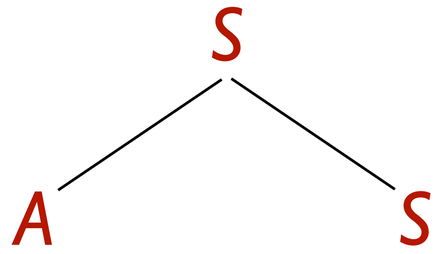
$\Rightarrow aabbb\underline{S}$

$\Rightarrow aabbb$



$\underline{S}$

$\Rightarrow \underline{A}S$



Ambiguous grammar: Parse trees

$S \rightarrow AS \mid \varepsilon$

$A \rightarrow Ab \mid aAb \mid ab$

$\underline{S}$

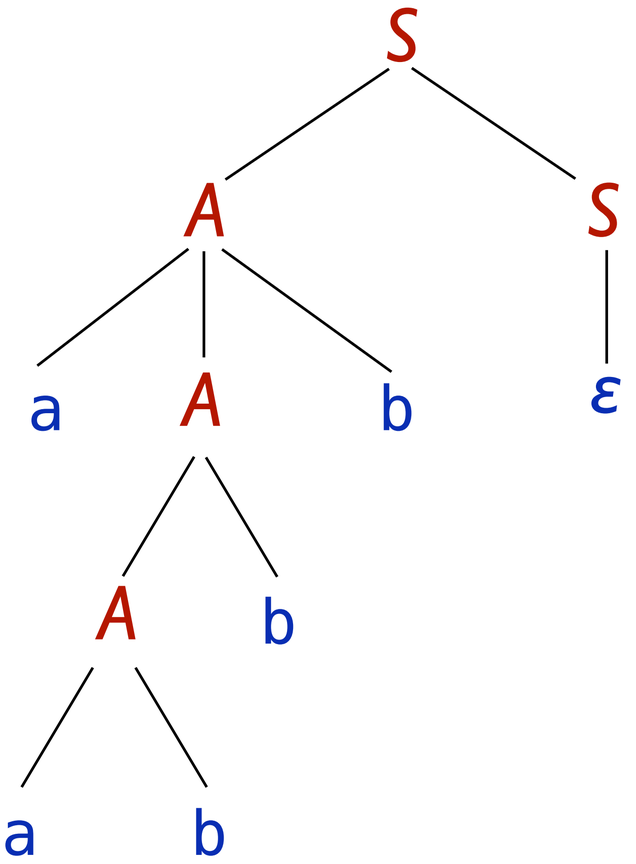
$\Rightarrow \underline{A}S$

$\Rightarrow aA\underline{b}S$

$\Rightarrow a\underline{A}bbS$

$\Rightarrow aabbb\underline{S}$

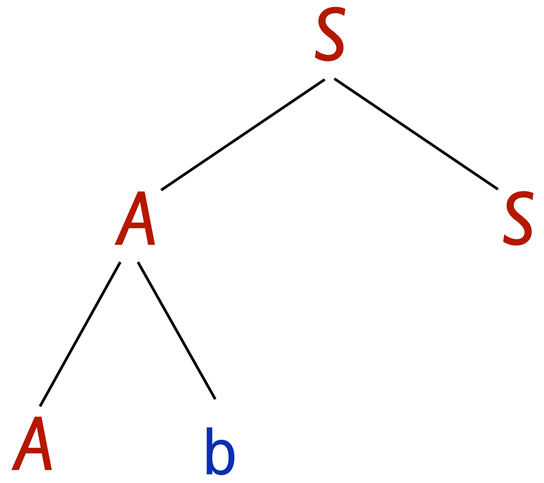
$\Rightarrow aabbb$



$\underline{S}$

$\Rightarrow \underline{A}S$

$\Rightarrow A\underline{b}S$



Ambiguous grammar: Parse trees

$S \rightarrow AS \mid \varepsilon$

$A \rightarrow Ab \mid aAb \mid ab$

$\underline{S}$

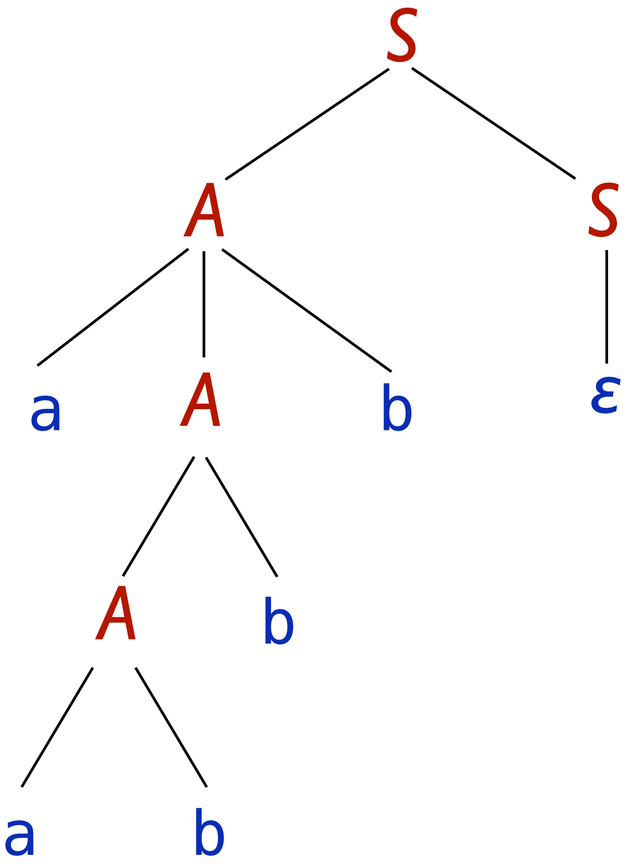
$\Rightarrow \underline{A}S$

$\Rightarrow aA\underline{b}S$

$\Rightarrow a\underline{A}bbS$

$\Rightarrow aabbb\underline{S}$

$\Rightarrow aabbb$

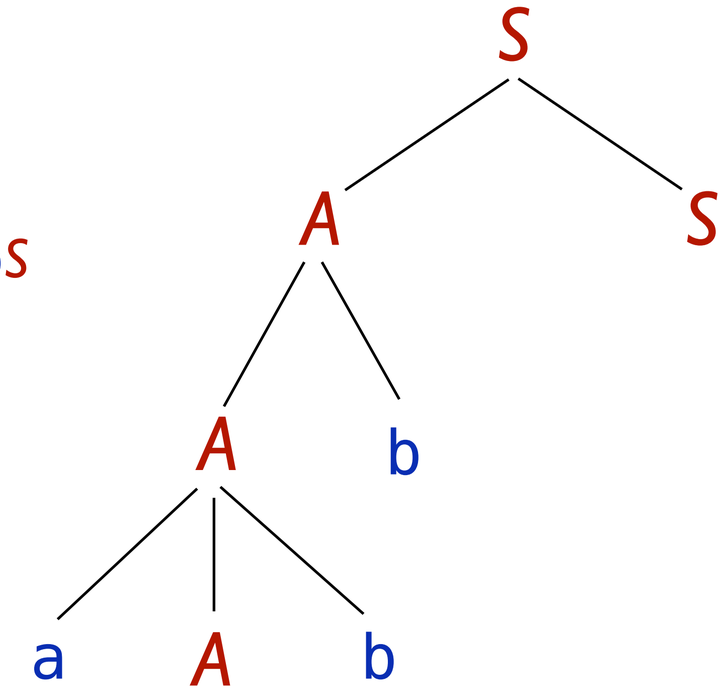


$\underline{S}$

$\Rightarrow \underline{A}S$

$\Rightarrow A\underline{b}S$

$\Rightarrow a\underline{A}bbS$



Ambiguous grammar: Parse trees

$S \rightarrow AS \mid \varepsilon$

$A \rightarrow Ab \mid aAb \mid ab$

$\underline{S}$

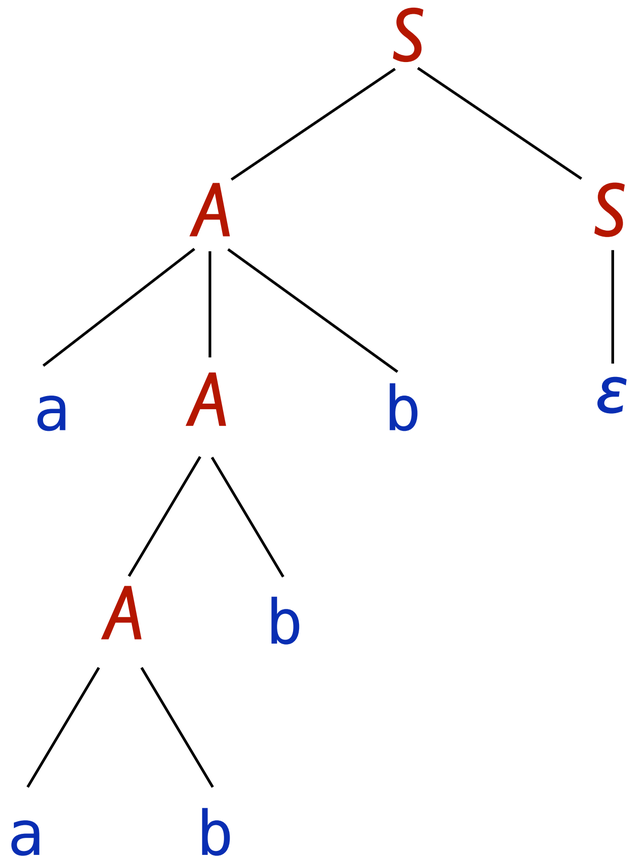
$\Rightarrow \underline{A}S$

$\Rightarrow aA\underline{b}S$

$\Rightarrow a\underline{A}bbS$

$\Rightarrow aabbb\underline{S}$

$\Rightarrow aabbb$



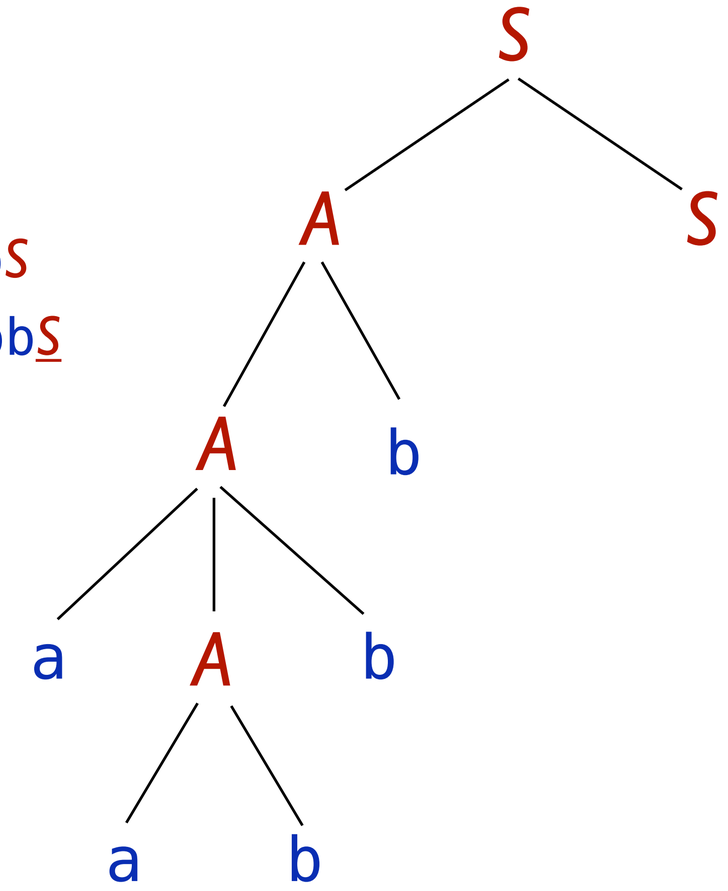
$\underline{S}$

$\Rightarrow \underline{A}S$

$\Rightarrow A\underline{b}S$

$\Rightarrow a\underline{A}bbS$

$\Rightarrow aabbb\underline{S}$



Ambiguous grammar: Parse trees

$S \rightarrow AS \mid \varepsilon$

$A \rightarrow Ab \mid aAb \mid ab$

$\underline{S}$

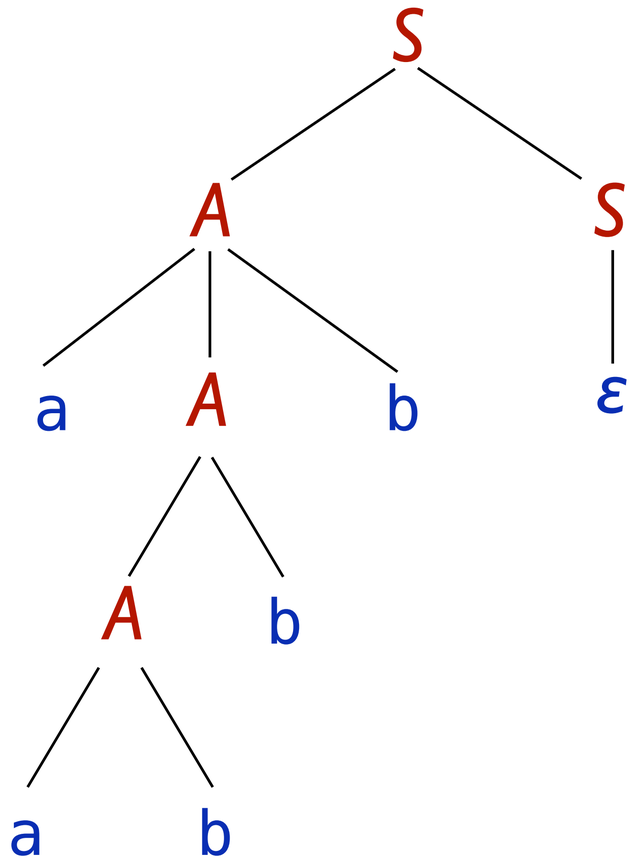
$\Rightarrow \underline{A}S$

$\Rightarrow a\underline{A}bS$

$\Rightarrow aA\underline{b}bS$

$\Rightarrow aabbb\underline{S}$

$\Rightarrow aabbb$



$\underline{S}$

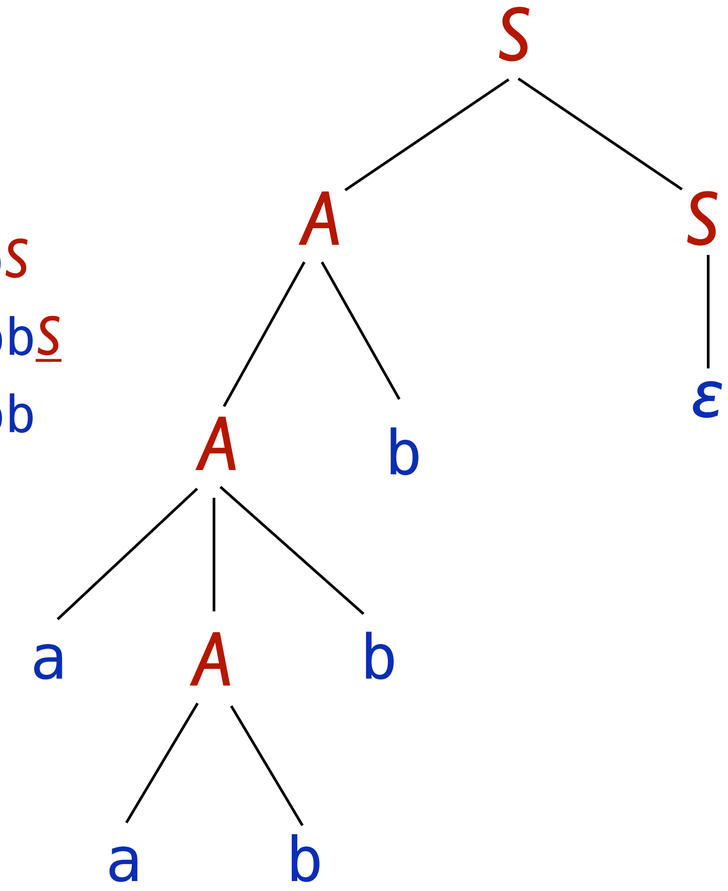
$\Rightarrow \underline{A}S$

$\Rightarrow \underline{A}bS$

$\Rightarrow aA\underline{b}bS$

$\Rightarrow aabbb\underline{S}$

$\Rightarrow aabbb$



Ambiguous grammar: Leftmost derivations

$S \to AS \mid \varepsilon$

$A \to Ab \mid aAb \mid ab$

The string aabbb also has the following two leftmost derivations from S:

$S \Rightarrow AS \Rightarrow aAbS \Rightarrow aAbbS \Rightarrow aabbbS \Rightarrow aabbb$

$S \Rightarrow AS \Rightarrow AbS \Rightarrow aAbbS \Rightarrow aabbbS \Rightarrow aabbb$

We can use $A \to Ab$ first or second to generate the extra b.

Ambiguity is a property of *grammars*, not *languages*.

There can be multiple grammars for the same language, where some are ambiguous and some aren't.

However, some languages are *inherently ambiguous*.

A CFL L is *inherently ambiguous* if every CFG for L is ambiguous.

See Problem 2.29 in Sipser for an example of an inherently ambiguous language.

The language of our example grammar isn't *inherently* ambiguous, even though the grammar *is* ambiguous.

To make G unambiguous, change the grammar to force the extra b's to be generated last, as shown below:

$$S \to AS \mid \varepsilon$$
$$A \to aAb \mid Ab \mid ab$$

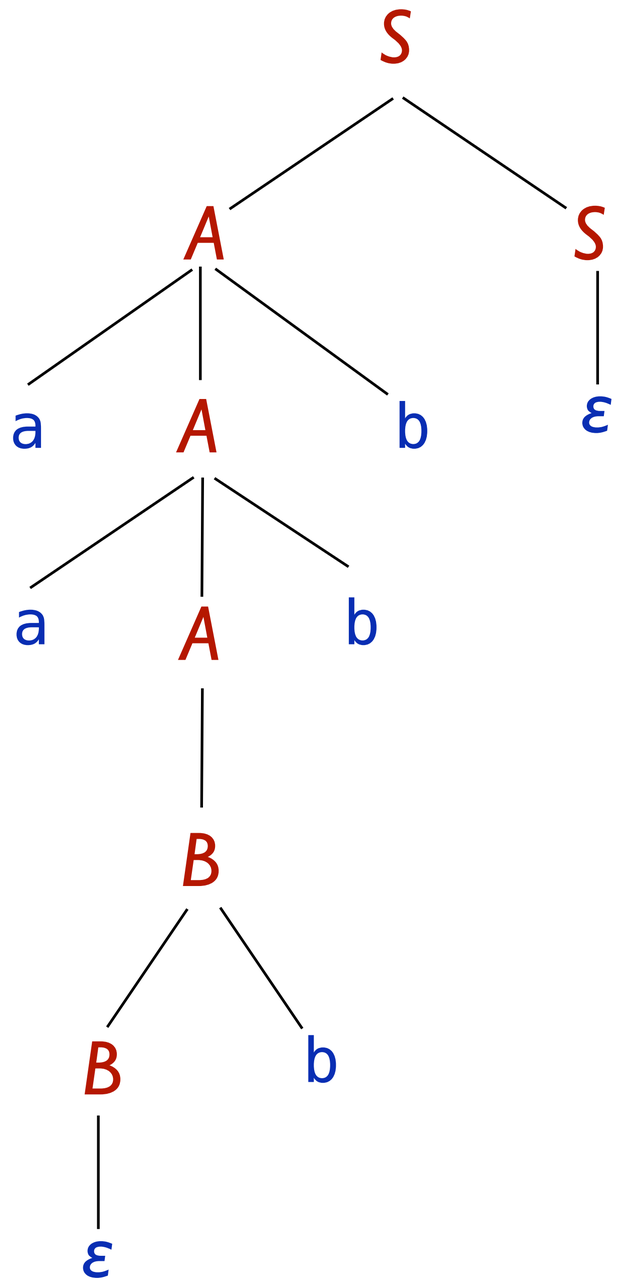
$$S \to AS \mid \varepsilon$$
$$A \to aAb \mid B$$
$$B \to Bb \mid \varepsilon$$

$$
\begin{aligned}
S &\rightarrow AS \mid \varepsilon \\
A &\rightarrow aAb \mid B \\
B &\rightarrow Bb \mid \varepsilon
\end{aligned}
$$

Now this grammar only allows one parse tree for the string and, equivalently, one leftmost derivation.

$$
\begin{aligned}
S &\rightarrow AS \mid \varepsilon \\
A &\rightarrow aAb \mid B \\
B &\rightarrow Bb \mid \varepsilon
\end{aligned}
$$

Now this grammar only allows one parse tree for the string and, equivalently, one leftmost derivation.



$$
\begin{aligned}
&\underline{S} \\
\Rightarrow\ &\underline{A}S \\
\Rightarrow\ &a\underline{A}bS \\
\Rightarrow\ &aa\underline{A}bbS \\
\Rightarrow\ &aa\underline{B}bbS \\
\Rightarrow\ &aa\underline{B}bbbS \\
\Rightarrow\ &aabbb\underline{S} \\
\Rightarrow\ &aabbb
\end{aligned}
$$

Ambiguity of a grammar implies that at least some strings in its language have *different structures*.

Different structures imply different meanings!

An inherently ambiguous language would be absolutely unsuitable as a programming language.

Expression grammar (G_1)

$E \rightarrow E + E \mid E * E \mid (E) \mid a$

Expression grammar (G_1)

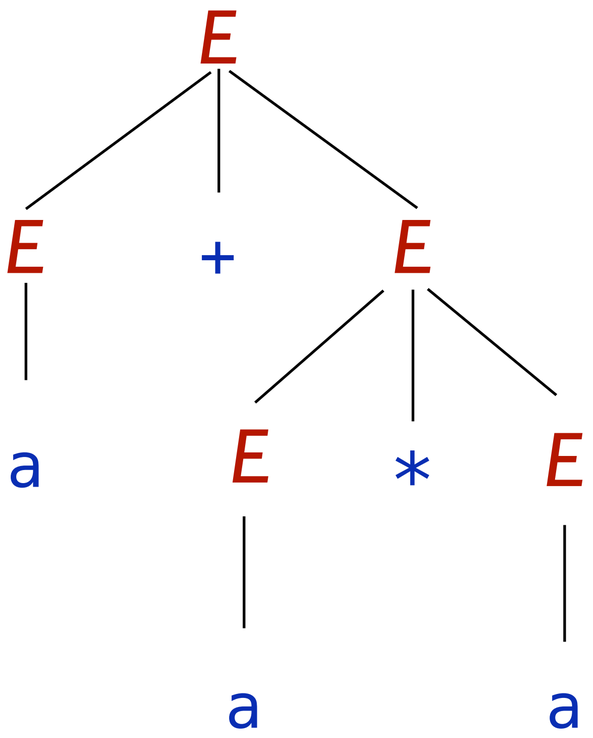
$E \rightarrow E + E \mid E * E \mid (E) \mid a$

The string a + a * a has two possible parse trees:

Expression grammar (G_1)

$E \rightarrow E + E \mid E * E \mid (E) \mid a$

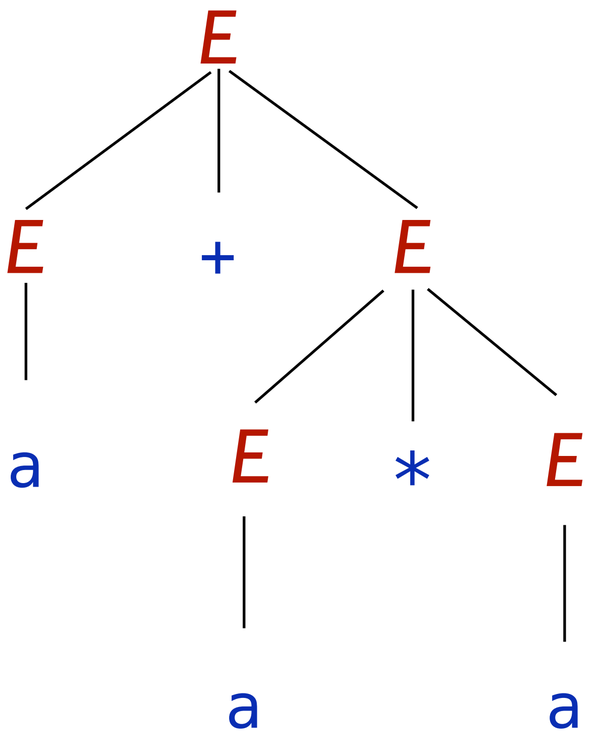
The string a + a * a has two possible parse trees:

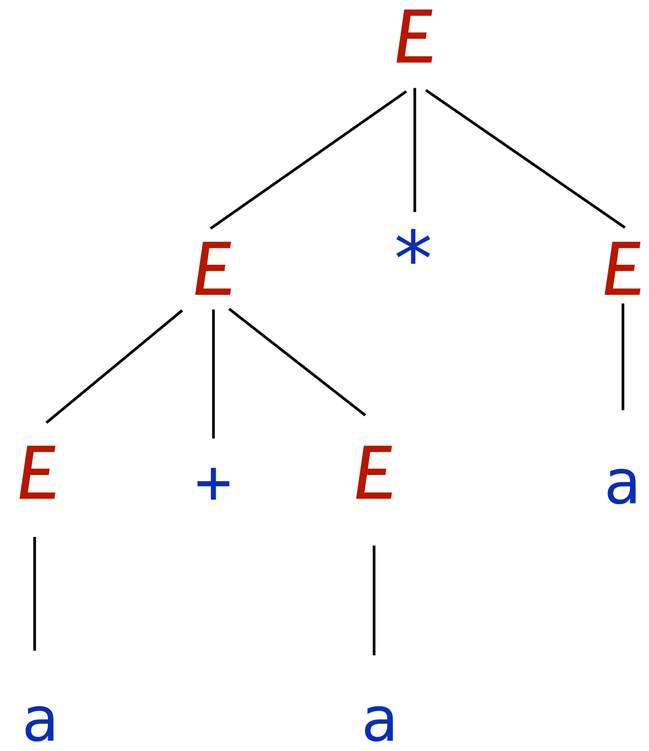


Expression grammar (G_1)

$E \rightarrow E + E \mid E * E \mid (E) \mid a$

The string a + a * a has two possible parse trees:

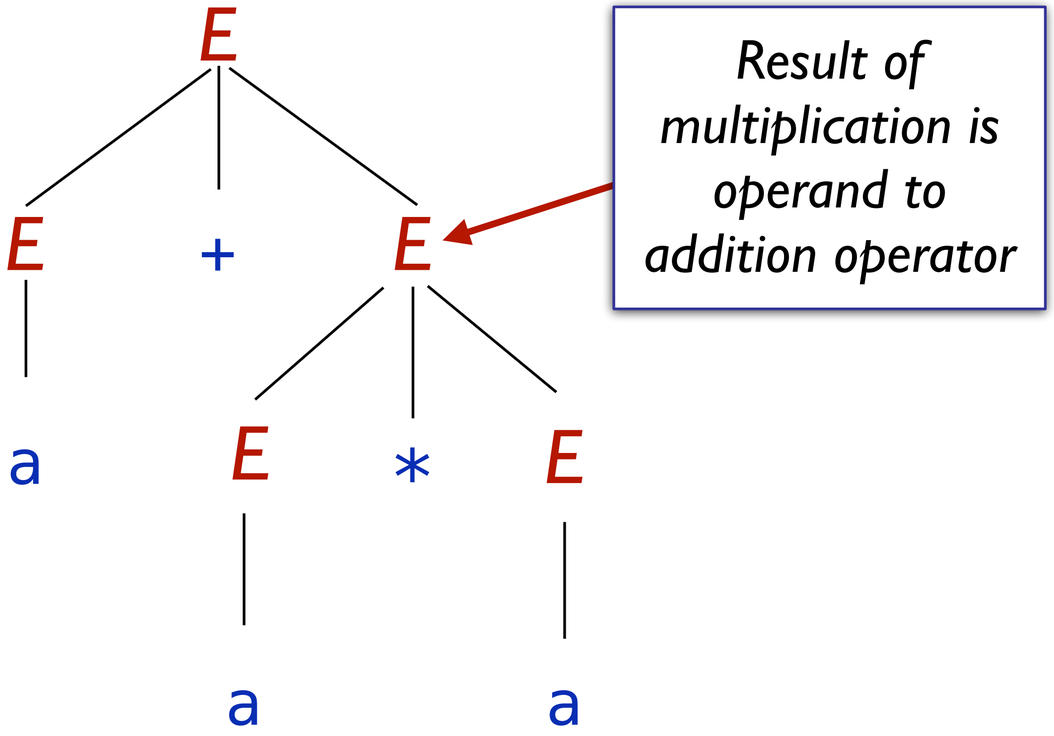


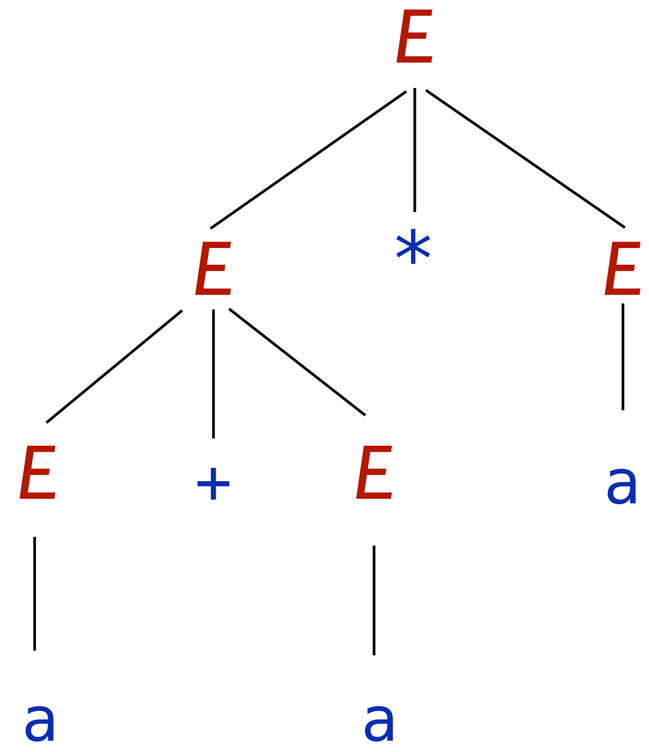


Expression grammar (G_1)

$E \rightarrow E + E \mid E * E \mid (E) \mid a$

The string a + a * a has two possible parse trees:

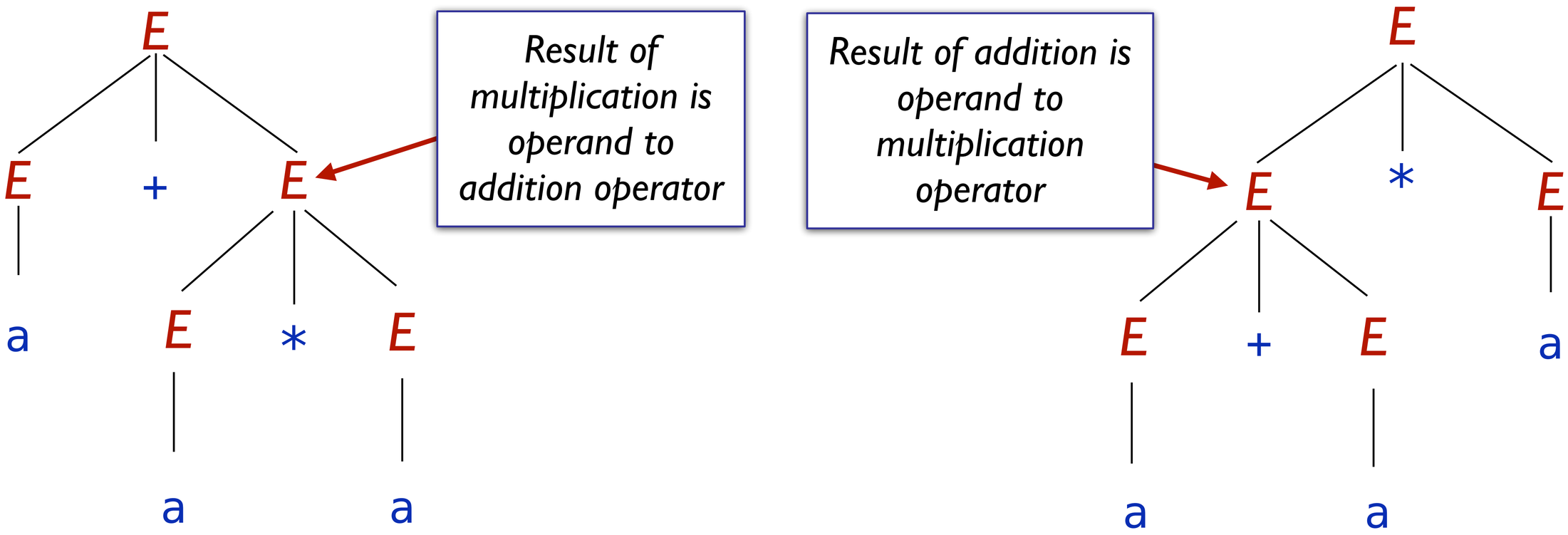




Expression grammar (G_1)

$E \rightarrow E + E \mid E * E \mid (E) \mid a$

The string a + a * a has two possible parse trees:



Removal of ambiguity

G_2 is an unambiguous version of G_1:

$$E \rightarrow E + T \mid T$$
$$T \rightarrow T * F \mid F$$
$$F \rightarrow (E) \mid \mathtt{a}$$

The easiest grammars for arithmetic expressions are ambiguous and need to be replaced by more complex, unambiguous grammars.

See G_4 and G_5 in Sipser.

“Danging `else`” in C

What does

```
if (a) if (b) s1; else s2;
```

mean?

```
if (a) {
    if (b)
        s1;
    else
        s2;
}
```

```
if (a) {
    if (b)
        s1;
} else
    s2;
```

Removing the parentheses and using a `then` keyword, we have the rule

$$\begin{aligned}\langle \textit{stmt} \rangle \rightarrow\ & \texttt{if}\ \langle \textit{expr} \rangle\ \texttt{then}\ \langle \textit{stmt} \rangle \\ & \mid \texttt{if}\ \langle \textit{expr} \rangle\ \texttt{then}\ \langle \textit{stmt} \rangle\ \texttt{else}\ \langle \textit{stmt} \rangle \\ & \mid \langle \textit{other-stmt} \rangle\end{aligned}$$

which allows two derivations for

$$\texttt{if}\ \langle \textit{expr} \rangle\ \texttt{then}\ \ \texttt{if}\ \langle \textit{expr} \rangle\ \texttt{then}\ \langle \textit{stmt} \rangle\ \texttt{else}\ \langle \textit{stmt} \rangle$$

We can introduce new non-terminals to distinguish matched and unmatched `then`/`else`:

We can introduce new non-terminals to distinguish matched and unmatched `then`/`else`:

⟨*stmt*⟩ → ⟨*matched*⟩ | ⟨*unmatched*⟩

⟨*matched*⟩ → `if` ⟨*expr*⟩ `then` ⟨*matched*⟩ `else` ⟨*matched*⟩ |
⟨*other-stmt*⟩

⟨*unmatched*⟩ → `if` ⟨*expr*⟩ `then` ⟨*stmt*⟩ |
`if` ⟨*expr*⟩ `then` ⟨*matched*⟩ `else` ⟨*unmatched*⟩

Applications of context-free grammars

Programming languages

Given a CFG describing the structure of a programming language and an input program (a string), the compiler or interpreter needs to recover the parse tree.

The parse tree represents the structure of the program – what's declared where, how expressions nest, etc.

Block → *Stmt*
| `{` *Stmts* `}`

Stmts → ε
| *Stmt Stmts*

Stmt → *Expr*`;`
| `if (`*Expr*`)` *Block*
| `while (`*Expr*`)` *Block*
| `do` *Block* `while (`*Expr*`);`
| *Block*
| …

Expr → `var`
| `const`
| *Expr* + *Expr*
| *Expr* – *Expr*
| *Expr* = *Expr*
| …

```
{
    var = var * var;
    if (var)
        var = const;
    while (var) {
        var = var + const;
    }
}
```

Grammars in compilers

One of the key steps in a compiler is figuring out what a program "means".

This is usually done by defining a grammar showing the high-level structure of a programming language.

- There are certain classes of grammars (LL(1) grammars, LR(1) grammars, LALR(1) grammars, etc.) for which it's easy to figure out how a particular string was derived.
- Tools like **`yacc`** or **`bison`** automatically generate parsers from those grammars.

Curious to learn more? Take CMPU 331!

Natural languages

Natural language processing

By building context-free grammars for human languages and applying statistical inference, it's possible for a computer to recover the likely meaning of a sentence.

In fact, CFGs were first called *phrase-structure grammars* and were introduced by Noam Chomsky in his book *Syntactic Structures* (1957).

Curious to learn more? Take CMPU 366!

CFGs allow natural recursion among things like *nouns*, *verbs*, *prepositions*, and their phrases.

Noun phrases can appear inside verb phrases and prepositional phrases, and vice versa.

```
NP The cat
   PP in
      NP the hat
         PP with
            NP a feather ...
```

S → *NP VP*

NP → *Pronoun* | *ProperNoun* | *Det Nominal*

Nominal → *Nominal Noun* | *Noun*

VP → *Verb* | *Verb NP* | *Verb NP PP* | *Verb PP*

PP → *Preposition NP*

Pronoun → `I` | `you` | `he` | `she` | `they` | ...

ProperNoun → `Patti` | `Tina` | `Yoko` | `Aretha` | ...

Det → `the` | `a` | `an` | *ProperNoun* `'s` | ...

Noun → `cat` | `dog` | `computer` | ...

...

Where are we?

A language L is called a *context-free language* (or CFL) if there is a CFG G such that $L = L(G)$

Questions:

- How do we design context-free grammars for context-free languages?
- What languages are context-free?
- How are context-free and regular languages related?

Acknowledgments

This lecture incorporates material from:

Nancy Ide, Vassar College

Amy Liu, Stanford University

Keith Schwarz, Stanford University

Michael Sipser, *Introduction to the Theory of Computation*